FG-ATTN: LEVERAGING FINE-GRAINED SPARSITY IN DIFFUSION TRANSFORMERS

Sankeerth Durvasula [1 2 3] **Kavya Sreedhar** [4] **Zain Moustafa** [1] **Suraj Kothawade** [4] **Ashish Gondimalla** [5]
Suvinay Subramanian [4] **Narges Shahidi** [* 4] **Nandita Vijaykumar** [* 1 2]

ABSTRACT

Generating realistic videos with diffusion transformers demands significant computation, with attention layers becoming the central bottleneck. Even producing a short clip requires running a transformer over a very long sequence of embeddings, e.g., more than 30K embeddings for a 5-second video. These long sequence lengths thus incur significant compute latencies. Prior work aims to mitigate this bottleneck by exploiting sparsity in the attention layers to reduce the computation required. However, these works typically rely on block-sparse attention, which skips score computation only when all entries in a *block* of attention scores (corresponding to M queries and M keys, with $M = 64$ typically) are zero. This coarse-granular skipping of attention scores does not fully exploit sparsity in the attention map and leaves significant room for improvement.

In this work, we propose FG-Attn, a sparse attention mechanism for long-context diffusion transformers that leverages sparsity at a fine granularity. Unlike block-sparse attention, which skips entire $M \times M$ blocks, our approach skips computations at the granularity of $M \times 1$ *slices* of the attention map. Each slice is produced as a result of query-key dot products between a block of query vectors and a *single key*. To implement our proposed sparse attention mechanism, we construct a new highly efficient bulk-load operation called asynchronous-gather load. This load operation gathers a sparse set of relevant key-value vectors from memory and arranges them into packed tiles in the GPU's shared memory. In this manner, only a sparse set of keys relevant to those queries are loaded into shared memory when computing attention for a block of queries, in contrast to loading full blocks of key tokens in block-sparse attention. Our fine-grained sparse attention, applied to video diffusion models, achieves an average 1.15x (up to 1.29x) speedup for 5 second, 720p videos, and an average 1.13x (up to 1.20x) for 5 second, 480p videos on a single H100 GPU.

Code: https://github.com/sankeerth95/FG-Attn

1 INTRODUCTION

Media generation models in deep learning are highly effective at capturing complex data distributions, such as videos (Wan et al., 2025; Kong et al., 2024), audio (Kong et al., 2020), 3D models (Zhao et al., 2025), and images (Esser et al., 2024b;a; Chen et al., 2023). When trained on real-world data, these models can generate realistic content, representing a major advancement in synthetic media generation. They enable a wide range of applications, such as advanced video editing, intuitive 3D modeling, and the creation of immersive 3D environments.

Video generation is a popular media generation task powered by diffusion models, a family of deep learning approaches that iteratively refine random-noise inputs to produce a sample of a data distribution. Diffusion models learn the *score function* (Section 2.1) of a complex data distribution using a parameterized function. In video generation, the data distribution is the set of real-world videos, and the parameterized function is typically a vision transformer (ViT) (Dosovitskiy et al., 2020), referred to as a diffusion transformer (DiT) (Peebles & Xie, 2023).

Videos are generated by iteratively refining (or *denoising*) the latent space representation of the video. This is done in two steps: First, the latent-space representation of video frames, represented as a sequence of embedding vectors, are initialized with random noise. Each embedding vector typically corresponds to a latent representation of (H, W) patches of pixels across F frames of the video. Second, these embeddings are iteratively refined by the transformer. At each step, the transformer takes as input the current embedding sequence, and produces a cleaner sequence of vec-

[*]Joint mentorship. [1]Department of Computer Science, University of Toronto, Toronto, Canada [2]Vector Institute, Toronto, Canada [3]Sankeerth Durvasula was supported by an internship at Google [4]Google, Mountain View, USA [5]Google, Sunnyvale, USA. Correspondence to: Sankeerth Durvasula <sankeerth@cs.toronto.edu>.

tors that becomes the input for the next iteration. Through this repeated denoising process, the embeddings gradually converge to the latent representation of realistic video frames. Finally, the refined embedding sequence is decoded back into the pixel space to synthesize the video.

Even a short, low-resolution video requires a very large number of embedding vectors. For example, state-of-the-art video DiT models such as Wan 2.1 (Wan et al., 2025) encodes a 5-second video at 720p resolution and 16 fps using 74000 embeddings. On a single H100 GPU, the Wan 2.1 1.3B model requires 5 minutes of runtime for transformer evaluation, and the Wan 2.1 14B model requires over 25 minutes. A significant portion of this latency is attributed to computing the attention layers of the transformer (91% of the runtime for this example with Wan 2.1). Since the attention computation scales quadratically with embedding count, longer or higher resolution videos incur significantly larger processing latencies. For instance, generating a 10-second video at 720p and 16 fps requires roughly twice as many embeddings, resulting in about four times the runtime. Thus, as the embeddings increase, the runtime becomes increasingly dominated by computation in the attention layer. In Wan 2.1 (Wan et al., 2025) 1.3B, attention layers take 76% of the runtime to produce 49 frames, and 91% for 81 frames of video.

The inputs to attention layers, however, contain significant redundancies. As shown in Fig.1, computing only 20% of the attention scores per head still produces a valid video with no noticeable loss in quality. Note that the generated video frames are slightly different because the denoising process is sensitive to small perturbations in early denoising iterations, and the diffusion model with sparse attention scores still captures the same video distribution.

Several works (Xu et al., 2025; Li et al., 2025; Yang et al., 2025; Xi et al., 2025; Zhang et al., 2025b) exploit this redundancy to accelerate video generation. Recent methods such as RadialAttention (Li et al., 2025), X-attention (Xu et al., 2025), and SparseVideoGen (Xi et al., 2025) observe that the most important attention scores are typically concentrated in static regions of the attention map, particularly around the diagonal (Section 2.2) across all attention layers. They propose using static *masks* for attention layers, requiring computing only a fraction of the scores. This makes attention computation more efficient and speeds up video generation. However, because these methods skip a fixed subset of attention scores at each head, they may also omit essential scores. To address this limitation, another line of work, e.g., Video Sparse Attention (VSA) (Zhang et al., 2025b), proposes to infer the attention mask dynamically at runtime, and use this mask in block sparse attention. They introduce additional parameters to the model, enabling it to learn the attention mask (i.e., the set of attention scores to compute).

Both the static and learned mask approaches described above typically rely on block sparse attention (Section 3.2) as the underlying sparse attention mechanism. Block sparse attention implementations in literature (Guo et al., 2024; Ye et al., 2025; Hong et al., 2023; Dao et al., 2022; Wang et al., 2024; Dong et al., 2024) skip computing attention scores between all pairs of M query tokens and M key tokens, where M is a hardware-dependent parameter, at each attention head. In practice, this often means skipping entire 64×64 blocks of attention scores, where $M = 64$ to match underlying GPU hardware parameters for efficient GPU implementations. While this approach provides some acceleration, it is only effective if all scores in the skipped block are guaranteed to be near zero.

We instead propose to leverage sparsity at a finer granularity which would offer much greater opportunity for reducing computation. For instance, skipping 16×16 blocks of attention can reduce the FLOP count by up to 70%, compared to only about 15% when using 64×64 blocks, without noticeably affecting video generation quality (see Section 3.2). Similarly, skipping 128×1 slices of query-key dot products (at bfloat16 precision) can reduce FLOPs by as much as 55%. These findings suggest that exploiting fine-grained sparsity in attention computation can yield substantially greater speedups than current coarse-grained block sparse implementations.

Figure 1. Left: A video frame from Wan 2.1 (Wan et al., 2025) for the prompt "horse bending and drinking water from a lake". Right: The same model generates a similar video using only 20% of attention scores per head, achieving comparable results with a fraction of the FLOPs.

In this work, we aim to develop a fine-grained sparse attention mechanism to accelerate diffusion model inference. Our goals are to: (1) design an efficient sparse attention mechanism that skips computing attention scores for faster runtimes *at a finer granularity*, i.e, skip computing smaller sets of attention scores at a time, compared to block sparse attention, and (2) propose a mechanism to identify which sets of attention scores need to be computed or skipped, i.e, determine a sparse attention mask for our attention mechanism.

To this end, we introduce FG-Attn, an efficient fine-grained sparse attention mechanism for diffusion transformer models. The key idea is to skip computing scores corresponding

to *slices* of size $M \times 1$ of the attention map, instead of the $M \times M$ blocks as in current sparse attention implementations. This results in skipping computation for attention scores produced by one key and a group of M contiguous queries. Such fine-grained skipping allows for discarding a larger number of insignificant attention score computations. The key challenges in achieving this are: (i) Mapping our sliced sparse attention strategy onto modern GPUs while incurring negligible overhead and maintaining high device utilization. This is difficult because naive implementations of fine-grained sparsity may incur irregular memory access and data movement, which can drastically reduce GPU hardware utilization. (ii) Computing a fine-grained attention mask that reliably identifies slices of attention scores which can be safely skipped.

For (i), efficient GPU attention implementations (FlashAttention (Dao et al., 2022)) typically load a block (usually a multiple of 64×64) of queries and keys from high-bandwidth memory (HBM) into on-chip SRAM before using tensor cores to compute pairwise query–key dot products. To efficiently skip insignificant slices of attention scores, however, we must load only the sparse set of relevant keys and values for a group of queries from HBM into SRAM. To make effective use of tensor cores, these loaded keys and values must be packed into SRAM in an appropriate format expected by the tensor core. To enable this, we construct a new load primitive, which we call the asynchronous gather-load, that loads a sparse set of key/value vectors into an SRAM in the format required by the tensor core (as a tile, with a swizzled layout). Since modern GPUs lack hardware support (e.g., Tensor Memory Accelerator, TMA) for accelerated address generation to load a sparse set of vectors, we emulate the asynchronous gather-load using existing asynchronous load instructions on the H100 GPU.

For (ii), we propose two training-free strategies to identify the set of keys to load for a given group of queries. Our first strategy takes inspiration from prior works that apply caching techniques to accelerate diffusion model inference. We observe that, within each attention head, the query–key pairs yielding significant attention scores remain largely stable across denoising iterations. Leveraging this, in the first denoising iteration, we compute the full attention map, identify the significant slices of attention scores via thresholding, and then reuse this information in subsequent denoising iterations in our sparse attention mechanism.

A limitation of this approach is the HBM overhead required to store cached masks. To mitigate this overhead, we also introduce a second lightweight altenative strategy. For a group of queries $q_1, q_2, \ldots, q_M$, we only compute dot products with a key if the group's mean query, q_{mean}, is likely to yield a significant attention score. Thus, instead of evaluating all keys, we load only the top-k keys determined by their score against q_{mean} (Section 4.4). This heuristic is motivated by the observation that nearby embeddings tend to produce similar query distributions, allowing us to safely approximate with far fewer key computations.

We demonstrate that FG-Attn enables faster video generation without sacrificing the output quality of the video. On state-of-art text-to-video generation models, we show that FG-Attn speeds up the video generation time by up to $1.29\times$ ($1.13\times$ on average) compared to Flash Attention 3 (Shah et al., 2024) baselin. Our contributions are:

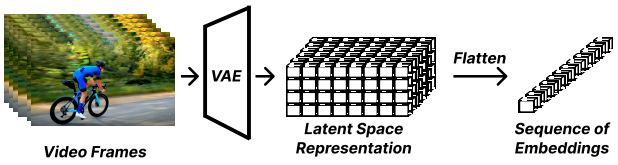
- We demonstrate that video diffusion models contain a significant amount of fine-grain sparsity in their attention maps that are not leveraged by existing block sparse attention methods.
- We introduce the first slice-based sparse attention mechanism that can practically exploit fine-grained sparsity on modern GPUs. To support this, we design a novel asynchronous gather-load primitive, which efficiently assembles sparse key/value vectors into tensor-core-compatible tiles, overcoming the overheads of irregular memory access.
- We demonstrate that sparsity patterns remain stable across denoising iterations, enabling a cache-based thresholding strategy that avoids recomputation while preserving accuracy.
- We propose two lightweight strategies for sparse mask generation that operate entirely without retraining. This ensures FG-Attn is directly applicable to existing state-of-the-art video DiTs.
- We show that our sliced attention mechanism can fully supersede existing block-sparse attention methods, achieving performance equivalent to or better than all prior coarse-grained approaches with negligible accuracy loss.

2 Background

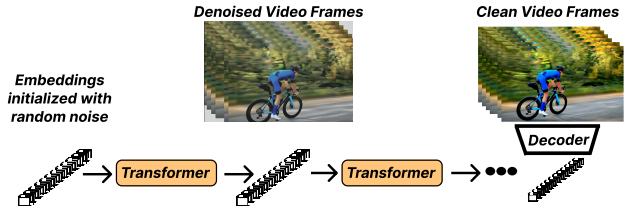
2.1 Video Diffusion Transformer Models

Diffusion models are a class of deep learning models that fit a score function, which is the gradient of the log probability of the data distribution x_d, i.e., $s(x_d) = \nabla_x log(p_{data})$ using a parameterized model. Latent space diffusion models fit the score function of the latent space representation of the data, denoted by x. The latent space representation consists of embedding vectors that can be decoded to or encoded from a data sample using an autoencoder. Fig. 2a shows how video can be encoded into a sequence of embedding vectors. Videos can be encoded into a set of high-dimension embedding vectors using a variational autoencoder model,

with each vector encoding information from a patch of pixels across a small set of frames. This set of embedding vectors can then be flattened to form a single sequence of embedding vectors. We use the terms "latent space diffusion models" and "diffusion models" interchangeably in this discussion.



(a) A set of noisy video frames are encoded into a sequence of latent space embeddings using a VQVAE.



(b) The diffusion transformer denoises the embeddings to generate clean video frame embeddings.

Figure 2. A latent space representation of video frames, represented as a set sequence of embeddings and encoded using a VQ-VAE, can be denoised using a diffusion transformer to produce embeddings corresponding to clean frames.

Generating a video sample from the diffusion model. Producing a video corresponds to drawing a sample from the fitted score function. This can be done by solving the probability flow Ordinary Differential Equation (ODE): $dx = s(x)dt$, where $s(x) = \nabla log(p_{data}(x))$ is the score function. This equation is solved numerically using integrators (Euler-Maruyama, Runge-Kutta, or specialized DPM Solvers (Lu et al., 2022)). These integrators iteratively evaluate the score function to find the direction of movement in the data space and update the latent space representation of the data x accordingly. This iterative refinement is known as *denoising*. Fig. 2b shows how videos are generated by starting with a noisy vector. A denoising function, implemented here as a transformer, progressively refines this vector into embeddings that correspond to clearer video frames.

2.2 Attention Computation

For a set of N queries $\mathbf{q_1}, \mathbf{q_2}, ...\mathbf{q_N}$, N keys $\mathbf{k_1}, \mathbf{k_2}, ...\mathbf{k_N}$ and N values $\mathbf{v_1}, \mathbf{v_2}, ...\mathbf{v_N}$ of dimension D, the self-attention layer computes, at each attention head,

$$\mathbf{O} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{D}}\right)\mathbf{V} \tag{1}$$

Where $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are $N \times D$ matrices consisting of query, key, and value vectors, respectively. $\mathbf{O}$ is the output matrix of size $N \times D$. The time-complexity of computing attention grows quadratically with the sequence size N, as $O(N^2D)$, resulting from computing a 2D matrix $\mathbf{Q}\mathbf{K}^T$ matrix of size $N \times N$. This 2D matrix, computed from the query, key matrices $(\mathbf{Q}, \mathbf{K})$ followed by a softmax operation is referred to as the *attention map*. Each element of the attention map is called the *attention score*. The expression for the attention map and attention scores (indexed by i, j) is given by:

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{D}}\right) \qquad a_{ij} = \frac{e^{\mathbf{q}_i\mathbf{k}_j}}{\sum_{n=1}^{N} e^{\mathbf{q}_i\mathbf{k}_n}}$$

2.3 Flash Attention for Accelerators

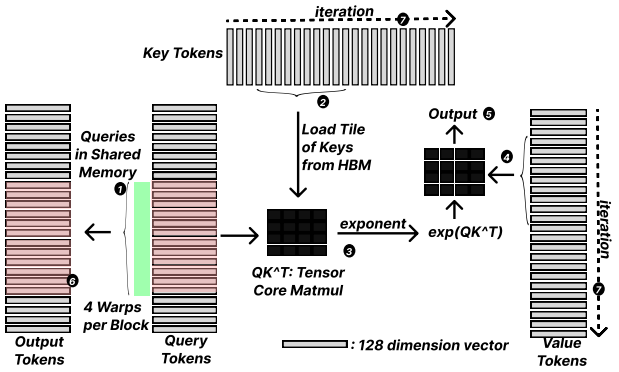


Figure 3. FlashAttention (Dao et al., 2022). Output tokens are computed by first loading a corresponding set of query tokens on chip. Then, the algorithm proceeds by iterating across all keys and corresponding values. A tile of keys and corresponding values are loaded on chip to compute attention scores, followed by exponentiation and multiplication with value tokens.

The memory footprint when computing attention naively is $O(N^2)$, where N is the sequence length. This is the result of computing and storing $\mathbf{Q}\mathbf{K}^T$ (Eq. 1), which requires materializing an $N \times N$ matrix for each attention head in memory. This becomes problematic when computing attention for long sequences, with the size of this intermediary $\mathbf{Q}\mathbf{K}^T$ matrix often exceeds the accelerator's HBM capacity.

Efficient implementations of attention in GPUs (flash attention (Dao et al., 2022)) avoids this high memory footprint by fusing the attention score computation and the multiplication of attention map with the value matrix. This involves computing tiles of $e^{\mathbf{Q}\mathbf{K}^T}$, storing them in on-chip memory instead of HBM, and multiplying them with the value vectors. This leads to an $O(N)$ memory footprint. As a result of less memory movement in storing the intermediate matrix in HBM, flash attention is less memory-bottlenecked and achieves high SM compute utilization (and thus, is fast).

Fig. 3 depicts how flash attention is mapped to a GPU kernel. The GPU kernel takes matrices for queries, keys, and values $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$ as input, and produces output matrix $\mathbf{O}$. The

queries, keys, values, and output tokens form a sequence of N tokens (gray bars), as shown in the figure. A set of contiguous output tokens for a particular head is computed within a threadblock/cooperative thread array. The figure shows a set of output tokens highlighted in green, computed by the threads of one threadblock. This set of tokens is labeled $\mathbf{O}_{tile}$ highlighted in red.

To compute $\mathbf{O}_{tile}$, the threadblock first loads a corresponding set of queries $\mathbf{Q}_{tile}$ into shared memory ❶. Then, the first set of key and value tokens, $\mathbf{K}_{tile}$ ❷ and $\mathbf{V}_{tile}$ ❹, is loaded into shared memory. An intermediary tile of keys/values (typically of size 64) is shown in the figure. This tile is used to compute intermediary $\mathbf{Q}_{tile}\mathbf{K}_{tile}^T$ using the tensor core ❸, followed by exponentiation to compute $e^{\mathbf{Q}_{tile}\mathbf{K}_{tile}^T}$. This intermediary result is stored in registers and is then multiplied by the corresponding $\mathbf{V}_{tile}$ using the tensor core ❺. The result of the computation is added to $\mathbf{O}_{reg}$, a slice of output in registers ❻. In the next iteration, the next tile $\mathbf{K}_{tile}$, $\mathbf{V}_{tile}$ in the sequence is loaded into shared memory, and computation is repeated ❼. The sum over the exponents $e^{\mathbf{Q}_{tile}\mathbf{K}_{tile}^T}$ is also computed in registers to hold the denominator of the softmax function (not shown in the figure).

3 Analysis

3.1 Latency of Video DiT Models

	49 frames	81 frames
Wan 1.3B, 480p	101s	204s
Wan 1.3B, 720p	354s	794s
Wan 14B, 480p	513s	1024s
Wan 14B, 720p	1727s	3823s

Table 1. Time to produce a video using a single H100 GPU chip.

As discussed in Section 2.1, video-DiT models first encode a set of video frames as latent embedding vectors using a vector-quantized variational autoencoder (VQVAE). The VQVAE compresses each spatiotemporal patch of $H \times W$ pixels across F consecutive frames into a single latent vector. For instance, setting $H = W = 8$ and $F = 4$ maps every 8×8 pixel block over 4 frames into one latent embedding vector. Even a short video, such as a five-second video at 480×832 resolution, gets encoded as approximately 32000 embedding vectors. Attention layers over such a large number of embeddings require substantial computation, as each layer must process an massive number of floating-point operations to generate even a short video.

Table 1 shows the amount of time required to produce a 5s video at 720p using the Wan 2.1 1.3B and 14B models (Wan et al., 2025) on a single H100 GPU chip. We see that a significant amount of time, about 10 minutes using Wan 14B model, is required to produce even a short video. In Fig. 4, we depict a breakdown of the time required by

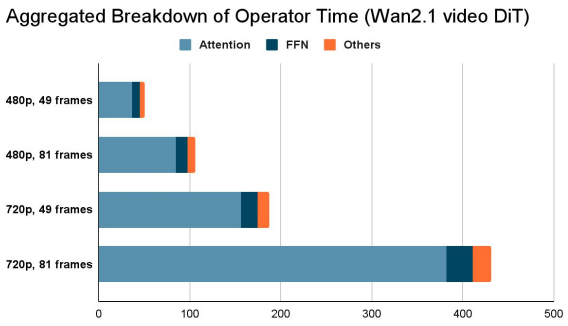


Figure 4. Breakdown of time spent (in seconds) by different operations during inference of Wan 2.1 1.3B (Wan et al., 2025) (eager mode).

each operator to produce the video (note that "others" here indicate the operators to encode the text tokens and the initial noisy video frames using the VQVAE). We observe that the majority of computation time is spent evaluating the transformer model, with the attention layer accounting for most of this cost. Specifically, when processing long sequences of embeddings (of length N), the attention operation scales as $O(N^2)$, in contrast to components such as the feed-forward network, which scale as $O(N)$. Consequently, for video-DiT models, longer sequences—arising from higher-resolution or longer videos—lead to an even greater fraction of time being dominated by attention. For example, in Wan 2.1 1.3B (Wan et al., 2025), nearly 91% of the runtime is spent computing attention when generating 81 frames at 720p, compared to the already-high 76% for 49 frames at 480p.

3.2 Attention Sparsity

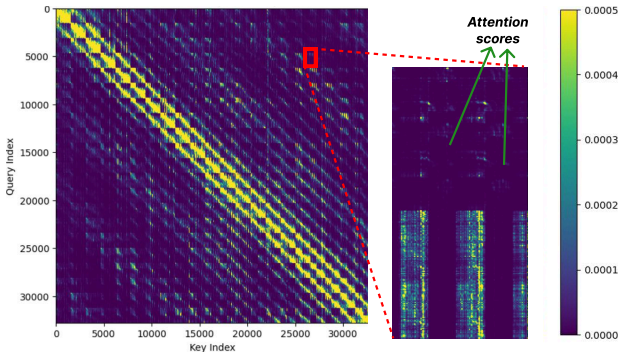


Figure 5. Sparsity in attention computation: attention scores are highly sparse, and locations of negligible attention scores are irregularly distributed.

Fig. 5 shows heatmap of the attention scores in one attention

heads of the Wan 2.1 1.3B vDIT model (Wan et al., 2025). We observe that the vast majority of attention scores are close to zero. Since many q, k pairs produce negligible attention scores, attention computation can be significantly accelerated by bypassing these score computations.

Prior sparse attention mechanisms such as FlexAttention (Dong et al., 2024), block-sparse attention (Guo et al., 2024) and FlashAttention (Dao et al., 2022) skip attention score computations to enable faster execution. In practice, these methods avoid loading and computing pairwise $q - k$ dot-products between contiguous sets (blocks) of q, k vectors (typically 64 queries or 64 keys). The block size M, or the number of queries/keys to skip computing attention score over is determined by the matrix multiplication operation dimensions in the tensor core of the accelerator (e.g., 64×64 on H100 GPUs). This allows $M \times M$ query-dot product computation to be skipped, corresponding to attention scores computed from M queries and M keys. However, in order to maintain accuracy, computing a block of attention scores can be skipped only when *all* query–key pairs within the block produce negligible attention scores.

Table 2. Block sparsity in attention scores: percentage of $M \times M$ blocks of attention scores where all scores are below a threshold. The threshold is set to $0.5/N$, where N is the sequence length of the input to the transformer.

Block size	Sparsity
128×128	5.5%
64×64	22.8%
32×32	47.7%
16×16	70.7%

Sparsity in attention scores is fine-grained. Table 2 shows the sparsity of the attention map at different block sizes of attention scores. We find that skipping finer-grained blocks (16×16) yields about 70% sparsity, whereas coarser blocks (64×64) achieve only 22%. This suggests that operating at finer granularity provides a greater opportunity for speedup. However, existing block-sparse attention implementations typically skip computations only at coarse block sizes. Current block sparse attention mechanisms (Guo et al., 2024; Dong et al., 2024) are unable to leverage finer-grain sparsity (below 64×64). In this work, we aim to exploit the higher sparsity available in finer-grained blocks to design a more efficient block-sparse attention mechanism that substantially reduces FLOPs without sacrificing accuracy.

4 METHOD

4.1 Overview

In order to leverage fine-grained sparsity for accelerating the attention layers in diffusion transformers, we must (1) implement a fine-grained sparse attention kernel on modern GPUs that skips computing slices attention scores as shown in Fig. 6, and (2) identify the slices of the attention map (mask) that can be skipped without sacrificing accuracy. To this end, we introduce FG-Attn, an efficient fine-grained sparse attention mechanism for diffusion transformers. To implement FG-Attn, we propose two techniques to determine the attention mask (Section 4.4) that determines which attention score slices are to be skipped. This mask is then supplied to FG-Attn's sparse attention kernel as input to efficiently compute attention.

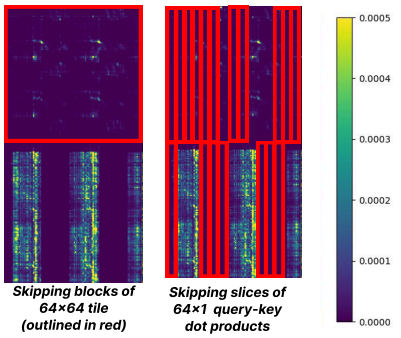


Figure 6. Block sparse attention mechanisms skip tiles of 128×128 attention map scores. We propose a method to skip fine-grain 128×1 sections of the attention scores.

4.2 Representing the Fine-grain Attention Mask

To implement fine-grained sparse attention, we need the attention mask to specify which slices of attention map scores to compute, i.e., the key/value vectors required for every group of M queries. For each group of M queries, we use a sequence of integers corresponding to the key/value vectors in the HBM. This sequence of indices is maintained in an array of integers representing the indices of keys/values to be loaded for each query group, for each attention head. Thus, for B batches, H heads, N/M groups of queries, and up to N keys, these indices are stored as a 4D integer array with dimensions $[B, H, N/M, N]$ in memory. M is selected to align with the unit dimension of the accelerator's matrix multiplication hardware (tensor cores), typically 64 or 128 in modern GPUs (with 128 used in our implementation on Hopper GPUs).

4.3 Efficient GPU Kernel Implementation

In FlashAttention (Dao et al., 2022), a set of M queries $\mathbf{Q}_{tile}$ and M contiguous keys $\mathbf{K}_{tile}$ are loaded into scratchpad memory before computing $\mathbf{Q}_{tile}\mathbf{K}_{tile}^T$ using the tensor core. On GPUs, these loads are efficiently handled by the Tensor Memory Accelerator (TMA). For FG-Attn, to compute query-key dot products over a sparse set of keys, we must instead *gather* a set of M non-contiguous keys (identified by the sparse-index mask) and pack them into a tile in

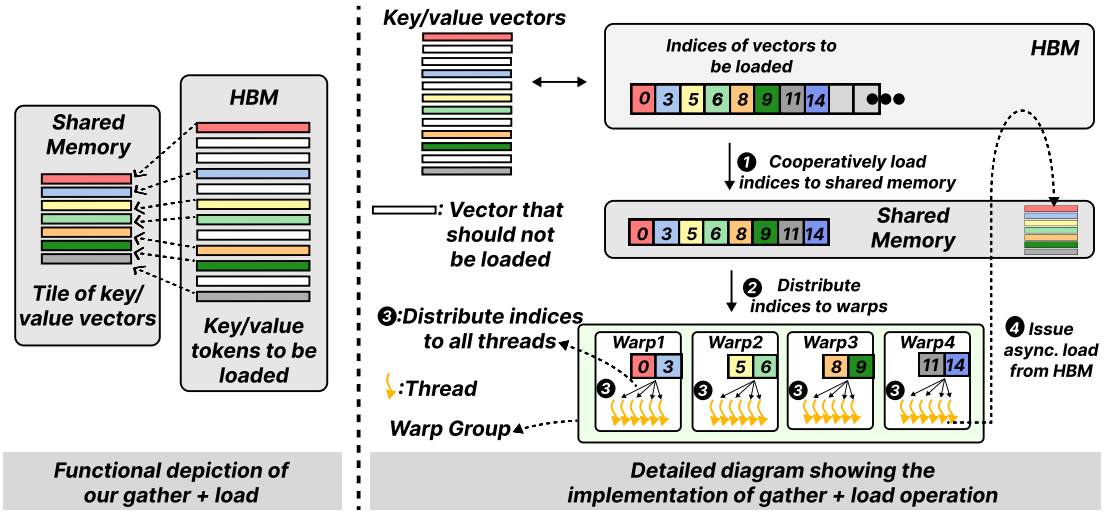


Figure 7. Loading a sparse set of keys/values using the sparse index mask into a tile in shared memory.

on-chip memory. We denote this relevant set of keys for a group of queries as $\mathbf{K}_{rel}$. The query tile and $\mathbf{K}_{rel}$ are then supplied to the tensor core to compute pairwise query-key dot products (i.e, $\mathbf{Q}_{tile}\mathbf{K}_{rel}$).

Loading a sparse set of key/value vectors from memory involves first generating (i.e, computing) the addresses of vector elements belonging to the vectors identified by the sparse index mask before issuing the load. To implement our attention mechanism efficiently on GPUs, we must ensure that (i) address generation for loading sparse key/value vectors corresponding to indices in the sparse index mask is fast, and (ii) the latency of address generation is hidden from the critical path to reduce overhead.

To achieve (i), we parallelize address generation by distributing the row indices used for deriving addresses across threads of a warp group. For (ii), we pipeline the address generation routine between loading a tile of key/value vectors and performing attention computation.

Parallelizing address generation with the sparse gather-load primitive. Given an array of vector indices, the gather-load primitive fetches the corresponding keys from HBM and assembles them in scratchpad as a contiguous tile. We incorporate this functionality as a device function, called the *gather-load* primitive. Fig. 7 (left) illustrates its functionality. For each group of M queries, we use the sparse-mask indices to load only the relevant keys onto the chip.

Figure 7 (right) shows an implementation overview of the gather-load primitive on H100 GPUs. The indices of the key/value vectors, stored in HBM, are provided as input to the sparse attention function. A set of these indices is cooperatively loaded into shared memory ❶, then distributed across the warps of each warp group (four warps per group on H100) ❷. Each warp's index set is then broadcast to all its threads using warp-shuffle instructions ❸. Finally, based on these row indices, each thread computes the addresses of its assigned vector elements and issues asynchronous load instructions to fetch the sparse vectors into shared memory ❹.

Overlap the address generation latency with attention computation. The latency of generating addresses for key/value vectors from the sparse index mask can be hidden behind attention computation. On NVIDIA H100 GPUs, the division of work of loading data and performing computation is organized between different warp groups: *producer* warp groups handle data loading, while the *consumer* warp group performs the computation. The producer warp group threads load tiles of query, key, and value vectors into on-chip memory, and the consumer warp group threads compute and accumulate the partial sums of output vectors.

An overview of the pipelining strategy is shown in Fig. 8. In each thread block, a M contiguous query vectors are first loaded into the scratchpad. Loading the relevant keys and values corresponding to these M queries is then performed across many iterations. address generation routine using the *gather-load* primitive function is then invoked (❶, ❷ and ❸). The producer threads load a set of indices corresponding to these M queries from the sparse index mask into shared memory ❶. Following this, the indices are loaded into registers ❷. These indices correspond to the rows of keys and values that need to be fetched. The producer then issues an asynchronous load operation to retrieve the corresponding rows from global memory ❸. The latency of the asynchronous load is fully hidden by the consumer warp's computation (attention score and output calculations) ❹. As a result, our implementation hides latency overhead due

to address generation from indirect indices for loading the relevant keys/values into memory.

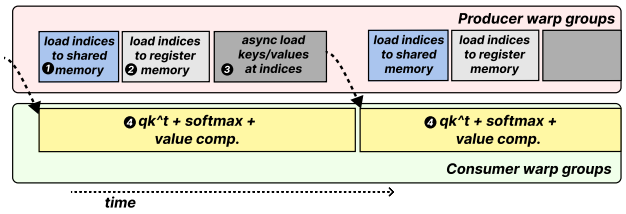


Figure 8. Pipelining gather-loading of keys/values into shared memory in the producer threads. The sparse index mask indices are loaded into registers first, followed by loading key/value tokens by the producer threads (*gather-load* primitive). The load latency of address generation is hidden by the attention computation.

End-to-end Implementation overview. An overview of our implementation is shown in Fig. 9. We build on FlashAttention (Dao et al., 2022), where each thread block computes a unique set of output vectors (shaded red among "Output Tokens" in Fig. 9). In each thread block, the producer warp groups first load M queries (typically 64 or 128) into scratchpad memory ❶. In the original FlashAttention (Dao et al., 2022) kernel, the producer warp groups then load contiguous blocks of M keys and compute attention scores as described in Section 2.3.

To implement FG-Attn, we modify this design by replacing the full key blocks with slices of M keys, determined by the sparse index mask and fetched using our gather-load primitive ❷. These sparse key tiles are supplied to the tensor core ❸ to compute only the relevant attention scores. The corresponding value vectors are loaded using the same sparse indices ❹, and a tensor core operation computes the linear combination of attention scores and value vectors to produce partial output sums ❺. These partial sums are accumulated into the output vectors ❻.

The process then advances to the next M slices of keys and values specified by the sparse index mask, and repeats until all relevant indices have been processed ❼. Across iterations, sparse key/value loading and tensor-core computation are pipelined.

4.4 Determining the Sparse-Index Mask

For a group of M queries $q_1, q_2, \ldots, q_M$, we determine whether a key k produces a *significant* attention coefficient with any of them. In this section, we present two training-free strategies for constructing the sparse attention mask.

Thresholding by caching attention mask across denoising iterations. Prior works (Hu et al., 2025; Ma et al., 2025) have observed that across denoising iterations, the intermediate embedding vectors remain approximately unchanged. Leveraging this observation, we cache the sparse

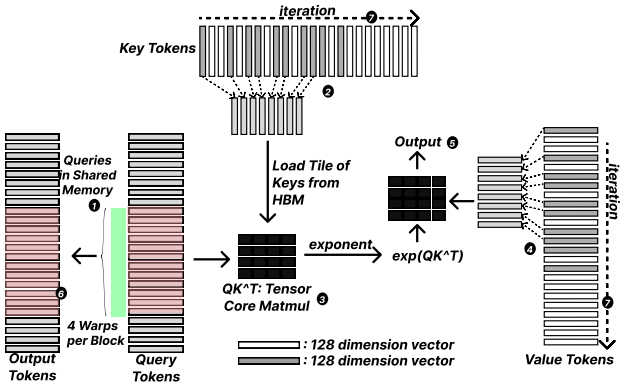


Figure 9. Implementation Overview. At each iteration, a sparse set of keys and values is loaded into the scratchpad. The load operation arranges the relevant keys into a tile in shared memory and uses tensor core operations to compute a partial sum of the output vector. The indices of keys and values to be loaded are supplied by the sparse index mask.

index mask derived from significant attention scores in an earlier iteration and reuse it in subsequent ones. Fig. 10 illustrates this approach. At denoising timestep t, the sparse attention mask for each attention head is derived from the attention scores that exceeded a predefined threshold in the preceding iteration $(t-1)$. In practice, we compute the

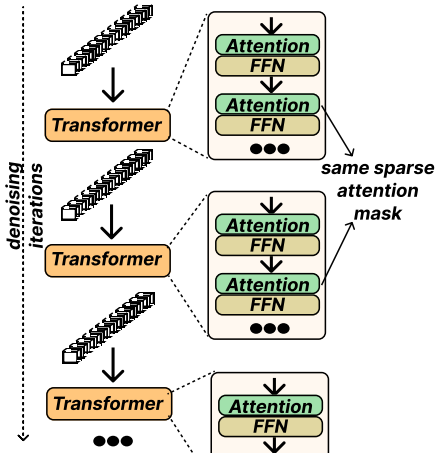


Figure 10. Determining which slices to attention mask based on attention scores observed in the previous denoising iteration.

sparse attention mask by evaluating the full attention scores and identifying slices with at least one score above a threshold τ_{cached}. The resulting mask is then stored in HBM for each attention head. In subsequent denoising iterations, this cached mask is reused by our sparse attention mechanism.

Thresholding based on average-query. In the context of video diffusion models, $q_1, q_2, \ldots, q_M$ are query tokens corresponding to adjacent pixels in space and time. Such ad-

jacent tokens typically exhibit similar responses compared to their surrounding queries. Motivated by this observation, we propose a simple, lightweight strategy for determining the attention mask. We compute the average of a group of queries,

$$q_{avg} = (q_1 + q_2 + .. + q_M)/M$$

A key k is included if its dot product with q_{avg} is significant. We then apply a threshold over these averaged scores. Specifically, the threshold τ is computed as:

$$\frac{\exp(k.q_{avg}/\sqrt{D})}{D} < \tau_1$$

4.5 Overheads

The sparse-index mask adds an additional $O(BHN^2/M)$ to the model's memory footprint. During attention computation, the producer warp groups load indices from the sparse-index mask onto the chip. This introduces a memory overhead, since the mask must reside in HBM. However, the latency of loading these indices can be hidden from the critical path, resulting in negligible runtime overhead.

5 Results

5.1 Methodology

We evaluate FG-Attn using the following open source, widely available video models: (1) Wan 2.1 (Wan et al., 2025) 1.3B, 14B models at 480p and 720p, at 81 frames, and (2) HunyuanVideo (Kong et al., 2024) at 720p. 81 frames. All experiments are conducted using bfloat16 precision. We implemented FG-Attn on top of FlashAttention-3 (Shah et al., 2024) for an H100 GPU. To evaluate the quality of the videos generated, we evaluate the quality of the videos generated using peak signal-to-noise ratio (PSNR) and structural similarity (SSIM), and measure their absolute video generation times. We generated videos for prompts from the Penguin Benchmark (Penguin Benchmark, 2024) and generate videos corresponding to the first 21 prompts in the benchmark for every video model. We also evaluate the videos we generated using the VBench (Huang et al., 2024) VLM benchmarking scores, alongside visual comparisons of frames from the generated videos in Section 5.3. We test two configurations of FG-Attn: one using the caching strategy to determine the mask (FG-Attn-cached), and the other using the pooling strategy (FG-Attn-pooling). For the FG-Attn-cached strategy, the threshold is set to $0.8/N$, where N is the number of embedding vectors in the latent space representation of the video. We compare FG-Attn with prior works that use block sparse attention to leverage sparsity in attention scores in DiTs: Radial Attention (Li et al., 2025), SparseVideo-Gen (Xi et al., 2025), SparseVideo-Gen2 (Yang et al., 2025), and SpargeAttn (Zhang et al., 2025a).

For a fair comparison with prior works, among all training-free acceleration methods, we set the number of "warmup" steps (initial steps running full attention) to be equal (24% of inference steps), similar to prior work (Yang et al., 2025). We note that for some open-source prior works, the baseline Flash Attention (FA) is different for warmup versus inference (e.g., in SVG2). Thus, we specify for each configuration which FA version is used for the two phases. We list the experiment configurations as follows:

- FA2-based Benchmarks: We compare against dense FA2 (Dense) (Dao, 2023), Radial Attention (Li et al., 2025), and SparseVideoGen (SVG) (Xi et al., 2025), all of which utilize FA2 for both warmup and sparse operations. We also include the open-source SparseVideoGen2 (SVG2) (Yang et al., 2025), which uses FA2 for warmup and FlashInfer's FA3 (Ye et al., 2025) attention kernel for sparse attention.
- FA3-based Benchmarks: We compare against dense FA3 (Shah et al., 2024), SpargeAttention (Zhang et al., 2025a)(FA3 warmup with FP8 attention), and a FlashInfer's FA3-based implementation of SVG2 (Yang et al., 2025) (utilizing FlashInfer). Ours is implemented entirely over FA3. We also test a hybrid variant, Ours + SVG2, which applies SVG2 clustering permutation to queries before executing our FA3-based attention.

5.2 Absolute Video Generation Times vs. Quality

In this section, we compare absolute wall clock time and video quality across various prior works. Tables 3, 4, 5 list the wall-clock time and PSNR/SSIM quality of the videos generated for the Wan-14B, Wan-1.3B and HunyuanVideo video model respectively. We make the following observations. First, on Wan-14B 720p, Wan 1.3B (720p/480p) and HunyuanVideo, FG-Attn outperforms SVG and Radial Attention in terms of video quality in the produced output video. The FA3 implementation of FG-Attn outperforms both the mechanisms. Second, SpargeAttn can generate videos much faster for every video model because it computes attention at lower precision (SpargeAttn for H100 uses 8-bit quantization, whereas SVG2/ours uses bf16) which compromises quality. Quantization is orthogonal to both FG-Attn and SVG2, and can be incorporated into these implementations as well. For HunyuanVideo, SpargeAttn produces significantly different videos when compared to the baseline, leading to significantly lower PSNR/SSIM. Third, FG-Attn achieves higher speedups compared to SVG2 (at 16-bit precision) on Wan 14B and Wan 1.3B at 480p resolution. While Sparse-VideoGen 2 outperforms FG-Attn at 720p resolution on HunyuanVideo, Wan 14B, and Wan

Table 3. Wall clock time and quality results on the Wan 14B model (Wan et al., 2025)

	Wan 14B 720p				Wan 14B 480p			
	PSNR	**SSIM**	**Time (mins)**	**FLOPs**	**PSNR**	**SSIM**	**Time (mins)**	**FLOPs**
Dense (FA2)	inf.	1.0	∼28	1X	inf.	1.0	7:30	1X
Radial (FA2+FA2)	21.1	0.72	20:45	0.48X	22.01	0.76	6:24	0.48X
SVG (FA2+FA2)	19.82	0.70	17:32	0.47X	20.13	0.74	5:18	0.47X
Dense (FA3)	inf.	1.0	16:45	1X	inf.	1.0	4:12	1X
Sparge (FA3+fp8)	22.51	0.79	13:04	-	20.11	0.78	3:58	-
SVG2 (FA3+FA3)	22.39	0.78	15:25	0.47X	23.48	0.84	4:58	0.47X
Ours (FA3+FA3)	23.01	0.79	15:55	0.68X	22.04	0.78	4:02	0.68X
Ours+SVG2 (FA3)	22.93	0.78	15:14	0.53X	22.31	0.78	4:01	0.53X

Table 4. Wall clock time and quality results on the Wan 1.3B model (Wan et al., 2025)

	Wan 1.3B 720p				Wan 1.3B 480p			
	PSNR	**SSIM**	**Time (mins)**	**FLOPs**	**PSNR**	**SSIM**	**Time (mins)**	**FLOPs**
Dense (FA2)	inf.	1.0	06:46	1X	inf.	1.0	1:38	1X
SVG (FA2+FA2)	22.40	0.78	4:01	0.47X	16.70	0.64	1:07	0.47X
Dense (FA3)	inf.	1.0	2:53	1X	inf.	1.0	1:01	1X
Sparge (FA3+f8)	27.10	0.84	2:26	-	21.41	0.79	00:43	-
SVG2 (FA3+FA3)	27.60	0.87	2:30	0.39X	22.44	0.82	1:00	0.39X
Ours (FA3+FA3)	27.03	0.85	2:14	0.59X	21.55	0.77	00:51	0.605X
Ours+SVG2 (FA3+FA3)	27.03	0.87	2:10	0.45X	21.61	0.77	00:58	0.44X

Table 5. Wall clock time and quality results on the HunyuanVideo model (Kong et al., 2024)

	HunyuanVideo 720p			
	PSNR	**SSIM**	**Time (mins)**	**FLOPs**
Dense (FA2)	inf.	1.0	12:38	1X
Radial (FA2)	22.33	0.77	09:07	0.48 X
SVG (FA2)	20.19	0.76	06:01	0.45X
Dense (FA3)	inf.	1.0	06:56	1X
Sparge (FA3+fp8)	-	-	05:04	-
SVG2 (FA3+FA3)	25.80	0.87	06:24	0.44X
Ours (FA3+FA3)	24.02	0.80	06:30	0.68X
Ours+SVG2 (FA3)	24.8	0.84	06:11	0.54X

1.3B, it does so by only a small margin, despite a much larger reduction in FLOPs. This is because SVG2 incurs significant overhead from computing query and key clusters, which becomes a larger portion of the total runtime on faster models. However, reordering query tokens into clusters before applying FG-Attn (denoted as Ours+SVG2) results in performance that exceeds SVG2 across all baselines.

5.3 Qualitative Analysis

Table 6 shows the VBench (Huang et al., 2024) video benchmarking results when compared to the baseline. We observe that FG-Attn achieves negligible degradation in quality when compared to the baseline. The attention mask is cached once every 15 DiT iterations for these experiments.

Figs. 12, 13 and 14 show the visual representation of the produced video compared to the original (the top row of each set of videos represents the baseline video) for the HunyuanVideo model, Wan 1.3B model, and the Wan 14B model, respectively. We find that across all the prompts tested here, FG-Attn can recover the original video with no quality degradation. FG-Attn also retains the generated video style and does not significantly shift the distribution captured by the underlying model.

Table 6. VBench quality metrics

	Aesthetic Quality	*Subject Consistency*	*Background Consistency*	*Overall Consistency*
Wan-1.3B 480p baseline	0.601	0.936	0.958	0.23
Wan-1.3B 480p FGAttn	0.605	0.939	0.96	0.23
Wan-1.3B 720p Baseline	0.61	0.944	0.962	0.233
Wan-1.3B 720p FGAttn	0.61	0.944	0.964	0.232
Wan-14B 480p baseline	0.623	0.953	0.97	0.25
Wan-14B 480p FGAttn	0.616	0.952	0.975	0.247
Wan-14B 720p baseline	0.621	0.945	0.969	0.248
Wan-14B 720p FGAttn	0.619	0.942	0.961	0.245
Hunyuan-13B 720p baseline	0.62	0.944	0.962	0.239
Hunyuan-13B 720p FGAttn	0.62	0.94	0.962	0.239

5.4 Ablation Study

Fig. 11 depicts the average attention computation time for video generation as the threshold parameter is varied. We sweep the threshold parameter from $0.1/N$ to $1/N$, where N is the number of embedding vectors in the latent space representation of the video. A higher threshold enables skipping a larger amount of computation, thereby leading to a speedup.

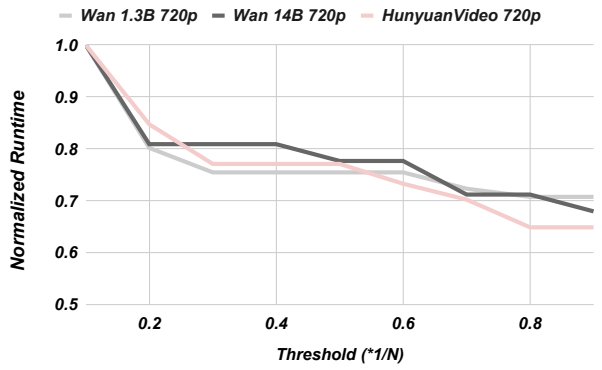


Figure 11. Normalized video generation time at different thresholds applied to FG-Attn-cached.

6 Related Work

Block sparse attention. Several implementations of block sparse attention (Guo et al., 2024; Dao et al., 2022; Dong et al., 2024; Ye et al., 2025; Wang et al., 2024) propose a coarse-grained sparse attention mechanism that skips entire blocks of attention score computations at granularity of 64×64 or 128×128 at half-precision. Current block-sparse attention mechanisms either prevent further reduction of block size (do not compile) or cause significant hardware underutilization and performance overhead, since they are constrained by the tensor core matrix multiplication width (Section 3.2). Several works in the large language model literature (Jiang et al., 2024; Xu et al., 2025; Hong et al., 2023; Yuan et al., 2025; Gao et al., 2024) utilize block sparse attention to accelerate attention computation.

Block sparse attention for videoDiTs. Recent works, such as Radial Attention (Li et al., 2025), X-attention (Xu et al., 2025), SparseVideoGen (Xi et al., 2025), and SparseVideoGen2 (Yang et al., 2025), have applied block sparse attention implementations to video diffusion models. These approaches consider a fixed sparsity pattern in the attention map based on empirical observations of significant patterns. Other works, such as Video Sparse Attention (Zhang et al., 2025b), incorporate learned sparse attention patterns by using a parameterized model to derive the attention map mask. Both approaches utilize coarse-grained sparse attention mechanisms. In contrast, our method enables fine-grained skipping of attention blocks, providing more opportunities for skipping computation. We compare FG-Attn with SparseVideoGen and Radial Attention in Section 5. Moreover, trainable sparse attention methods such as Video Sparse Attention (VSA) (Zhang et al., 2025b) can be reformulated to generate sparse masks compatible with FG-Attn 's attention kernel. These methods are orthogonal to FG-Attn 's kernel implementation and can be used in conjunction as mask-determination strategies for FG-Attn.

Other techniques to accelerate video diffusion. SpargeAttention (Zhang et al., 2025a), SageAttention (Zhang et al., 2024b), and SageAttention2 (Zhang et al., 2024a) propose general attention approximation techniques, such as quantization and token compression mechanisms, that can be applied during inference for both LLM and DiT models. Token compression-based approaches may skip essential tokens relevant to the video, which could lead to inconsistent video generation (pointed out by (Li et al., 2025)). These approaches are orthogonal to our FG-Attn.

Figure 12. Samples of videos generated using baseline HunyuanVideo model, and FG-Attn-HunyuanVideo (The baseline generates first row, second row generated using FG-Attn)

7 DISCUSSION

While this work primarily evaluates FG-Attn on video diffusion transformers (DiTs), the fine-grained sparse attention mechanism is broadly applicable to other long-context diffusion transformer models, such as 3D generative models, large language diffusion models, audio diffusion models, and 3D generative models, all of which exhibit redundancy in their attention maps. The slice-based skipping introduced in FG-Attn can be incorporated into these settings, making it a general drop-in replacement for block-sparse attention.

Our current design of FG-Attn is implemented in NVIDIA H100 GPU. However, the asynchronous gather-load primitive can be re-implemented to align with other accelerators' memory hierarchies and matrix multiplication units. For instance, TPUs with sparse core architectures can be readily used to implement our gather load primitive. Next-generation GPUs such as NVIDIA's B100 introduce hardware support for gathering tiles/vectors, which FG-Attn can readily take advantage of.

A limitation of FG-Attn is that the low overheads of the gather-load operation may be difficult to obtain when attention computation is not compute-bound, such as LLM decoding, as it may not be possible to fully overlap the latency of address generation and data loading.

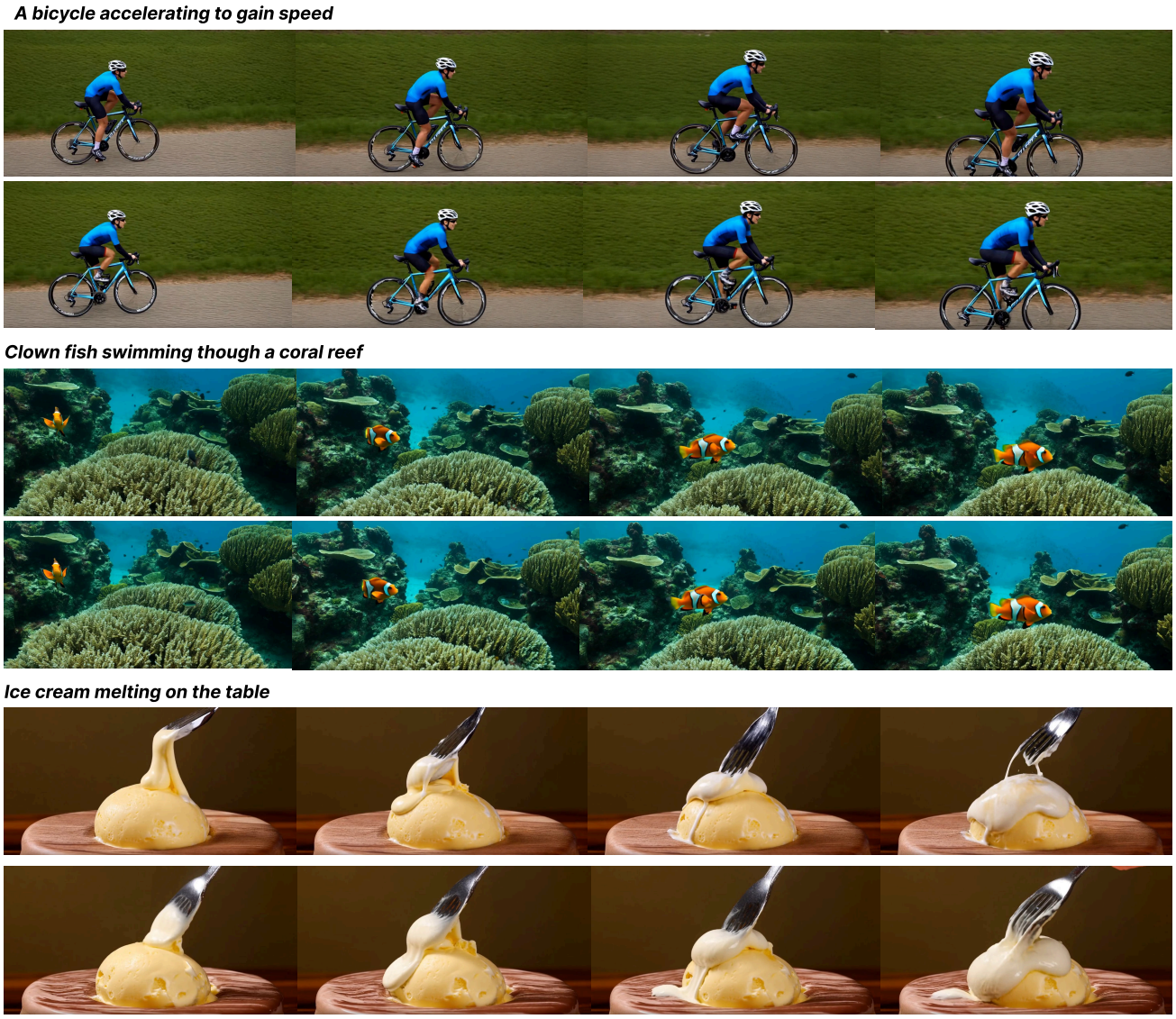


Figure 13. Samples of videos generated using baseline Wan-1.3B model, and FG-Attn-Wan1.3B. (First row is generated by the baseline, second row is generated using FG-Attn)

8 Conclusion

This paper presents FG-Attn, a novel fine-grain sparse attention mechanism. FG-Attn skips attention computations at a more granular, slice-based level ($M \times 1$) rather than large blocks ($M \times M$), as implemented by current block sparse attention mechanisms. We implement this using our asynchronous gather-load primitive that efficiently loads only the relevant, sparse key-value tokens into on-chip memory asynchronously without overheads. We proposed two training-free strategies to determine the sparse attention mask: one based on caching the mask across denoising iterations and another lightweight method using a query-averaging heuristic. Using FG-Attn, we demonstrate an average speedup of $1.48\times$, and up to $1.65\times$ on state of art video diffusion models.

9 Acknowledgement

This project was done as part of Sankeerth Durvasula'a internship at Google. I also want to acknowledge the contributions of Tianlei Pang from the University of Toronto for help with additional experiments.

References

Chen, J., Yu, J., Ge, C., Yao, L., Xie, E., Wu, Y., Wang, Z., Kwok, J., Luo, P., Lu, H., et al. Pixart-alpha: Fast training

Figure 14. Samples of videos generated using baseline Wan-14B model, and FG-Attn-Wan14B (First row is generated by the baseline, second row is generated by FG-Attn)

of diffusion transformer for photorealistic text-to-image synthesis. *arXiv preprint arXiv:2310.00426*, 2023.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Dong, J., Feng, B., Guessous, D., Liang, Y., and He, H. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Esser, P., Kulal, S., Blattmann, A., Entezari, R., Müller, J., Saini, H., Levi, Y., Lorenz, D., Sauer, A., Boesel, F., et al. Scaling rectified flow transformers for high-resolution image synthesis. In *Forty-first international conference on machine learning*, 2024a.

Esser, P., Townsend, M. S. M., Kulal, S., Dockhorn, T., Müller, J., Alterovych, A., Dehaerne, D., Lu, P. T. H., Hazirbas, C., Rampas, D., Rombach, R., D'Asaro, J., Watson, D., Voinea, D., Puzon, L., Boureau, Y.-L., and Mentzer, F. Flux: A unified approach to pixel-based and latent-space diffusion models, 2024b.

Gao, Y., Zeng, Z., Du, D., Cao, S., Zhou, P., Qi, J., Lai, J., So, H. K.-H., Cao, T., Yang, F., et al. Seerattention: Learning intrinsic sparse attention in your llms. *arXiv preprint arXiv:2410.13276*, 2024.

Guo, J., Tang, H., Yang, S., Zhang, Z., Liu, Z., and Han, S. Block Sparse Attention. https://github.com/mit-han-lab/Block-Sparse-Attention, 2024.

Hong, K., Dai, G., Xu, J., Mao, Q., Li, X., Liu, J., Chen, K., Dong, Y., and Wang, Y. Flashdecoding++: Faster large language model inference on gpus. *arXiv preprint arXiv:2311.01282*, 2023.

Hu, Z., Meng, J., Akhauri, Y., Abdelfattah, M. S., Seo, J.-s., Zhang, Z., and Gupta, U. Accelerating diffusion language model inference via efficient kv caching and guided diffusion. *arXiv preprint arXiv:2505.21467*, 2025.

Huang, Z., He, Y., Yu, J., Zhang, F., Si, C., Jiang, Y., Zhang, Y., Wu, T., Jin, Q., Chanpaisit, N., et al. Vbench: Comprehensive benchmark suite for video generative models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21807–21818, 2024.

Jiang, H., Li, Y., Zhang, C., Wu, Q., Luo, X., Ahn, S., Han, Z., Abdi, A. H., Li, D., Lin, C.-Y., et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515, 2024.

Kong, W., Tian, Q., Zhang, Z., Min, R., Dai, Z., Zhou, J., Xiong, J., Li, X., Wu, B., Zhang, J., et al. Hunyuanvideo: A systematic framework for large video generative models. *arXiv preprint arXiv:2412.03603*, 2024.

Kong, Z., Ping, W., Huang, J., Zhao, K., and Catanzaro, B. Diffwave: A versatile diffusion model for audio synthesis. *arXiv preprint arXiv:2009.09761*, 2020.

Li, X., Li, M., Cai, T., Xi, H., Yang, S., Lin, Y., Zhang, L., Yang, S., Hu, J., Peng, K., et al. Radial attention: $o(nlogn)$ sparse attention with energy decay for long video generation. *arXiv preprint arXiv:2506.19852*, 2025.

Lu, C., Zhou, Y., Bao, F., Chen, J., Li, C., and Zhu, J. Dpm-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps. *Advances in neural information processing systems*, 35:5775–5787, 2022.

Ma, X., Yu, R., Fang, G., and Wang, X. dkv-cache: The cache for diffusion language models. *arXiv preprint arXiv:2505.15781*, 2025.

Peebles, W. and Xie, S. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 4195–4205, 2023.

Penguin Benchmark. Penguin video benchmark. https://huggingface.co/datasets/a-r-r-o-w/penguin-video-benchmark, 2024. Accessed: 2025-11-27.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685, 2024.

Wan, T., Wang, A., Ai, B., Wen, B., Mao, C., Xie, C.-W., Chen, D., Yu, F., Zhao, H., Yang, J., et al. Wan: Open and advanced large-scale video generative models. *arXiv preprint arXiv:2503.20314*, 2025.

Wang, G., Zeng, J., Xiao, X., Wu, S., Yang, J., Zheng, L., Chen, Z., Bian, J., Yu, D., and Wang, H. Flashmask: Efficient and rich mask extension of flashattention. *arXiv preprint arXiv:2410.01359*, 2024.

Xi, H., Yang, S., Zhao, Y., Xu, C., Li, M., Li, X., Lin, Y., Cai, H., Zhang, J., Li, D., et al. Sparse videogen: Accelerating video diffusion transformers with spatial-temporal sparsity. *arXiv preprint arXiv:2502.01776*, 2025.

Xu, R., Xiao, G., Huang, H., Guo, J., and Han, S. Xattention: Block sparse attention with antidiagonal scoring. *arXiv preprint arXiv:2503.16428*, 2025.

Yang, S., Xi, H., Zhao, Y., Li, M., Zhang, J., Cai, H., Lin, Y., Li, X., Xu, C., Peng, K., et al. Sparse videogen2: Accelerate video generation with sparse attention via semantic-aware permutation. *arXiv preprint arXiv:2505.18875*, 2025.

Ye, Z., Chen, L., Lai, R., Lin, W., Zhang, Y., Wang, S., Chen, T., Kasikci, B., Grover, V., Krishnamurthy, A., et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

Yuan, J., Gao, H., Dai, D., Luo, J., Zhao, L., Zhang, Z., Xie, Z., Wei, Y., Wang, L., Xiao, Z., et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

Zhang, J., Huang, H., Zhang, P., Wei, J., Zhu, J., and Chen, J. Sageattention2: Efficient attention with thorough outlier smoothing and per-thread int4 quantization. *arXiv preprint arXiv:2411.10958*, 2024a.

Zhang, J., Wei, J., Huang, H., Zhang, P., Zhu, J., and Chen, J. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration. *arXiv preprint arXiv:2410.02367*, 2024b.

Zhang, J., Xiang, C., Huang, H., Xi, H., Zhu, J., Chen, J., et al. Spargeattention: Accurate and training-free sparse attention accelerating any model inference. In *Forty-second International Conference on Machine Learning*, 2025a.

Zhang, P., Huang, H., Chen, Y., Lin, W., Liu, Z., Stoica, I., Xing, E., and Zhang, H. Vsa: Faster video diffusion with trainable sparse attention. *arXiv preprint arXiv:2505.13389*, 2025b.

Zhao, Z., Lai, Z., Lin, Q., Zhao, Y., Liu, H., Yang, S., Feng, Y., Yang, M., Zhang, S., Yang, X., et al. Hunyuan3d 2.0: Scaling diffusion models for high resolution textured 3d assets generation. *arXiv preprint arXiv:2501.12202*, 2025.